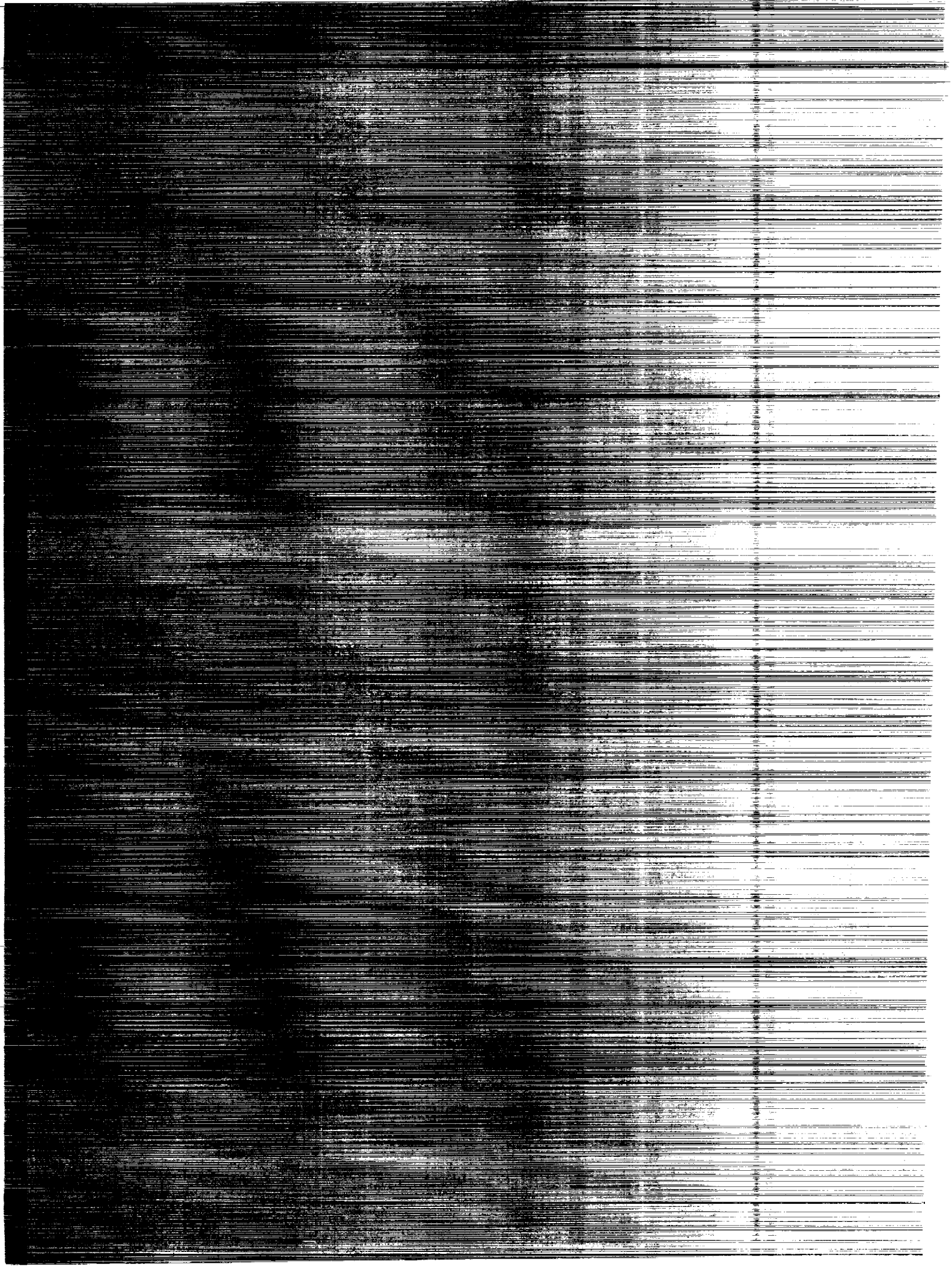
NASA Technical Memorandum 4305

Gigaflop Performance on a CRAY-2: Multitasking a Computational Fluid Dynamics Application

Geoffrey M. Tennille, Andrea L. Overman,
Jules J. Lambiotte, and Craig L. Streett

OCTOBER 1991



NASA Technical Memorandum 4305

Gigaflop Performance on a CRAY-2: Multitasking a Computational Fluid Dynamics Application

Geoffrey M. Tennille, Andrea L. Overman,
Jules J. Lambiotte, and Craig L. Streett
Langley Research Center
Hampton, Virginia

National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Program
1991

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Abstract

This paper describes the methodology for converting a large, long-running applications code that executed on a single processor of a CRAY-2 supercomputer to a version that executed efficiently on multiple processors. Although the conversion of every application is different, a discussion of the types of modifications used to achieve gigaflop performance is included to assist others in the parallelization of applications for CRAY computers, especially those that were developed for other computers. An existing application, from the discipline of computational fluid dynamics, that had utilized over 2000 hours of CPU (central processing unit) time on a CRAY-2 during the previous year was chosen as a test case to study the effectiveness of multitasking on a CRAY-2. The nature of the dominant calculations within the application indicated that a sustained computational rate of 1 billion floating-point operations per second, or 1 gigaflop, might be achievable. The code was first analyzed and modified for optimal performance on a single processor in a batch environment. After optimal performance on a single CPU was achieved, the code was modified to use multiple processors in a dedicated environment. The results of these two efforts were merged into a single code that had a sustained computational rate of over 1 gigaflop on a CRAY-2. Timings and analysis of performance are given for both single- and multiple-processor runs.

1. Introduction

This paper provides a methodology for other researchers who wish to use parallel processing on CRAY computers. Some of the transformations that were made were obvious, but performance improvements often exceeded expectations. The code chosen for this effort was a large, long-running computational fluid dynamics application. A complete execution of this application ran for over 2000 hours on a single processor of a CRAY-2 supercomputer. Most of the computational time for this application was contained within matrix multiplication and fast Fourier transform (FFT) subroutines. The fact that these routines can be run in parallel, coupled with the large, long-running nature of the job, made this code an obvious candidate for parallel execution.

The result of the conversion was a code that executed at 1.01 billion floating-point operations per second (or 1.01 gigaflops) on a four-processor CRAY-2 with static random-access memory (SRAM) at Langley Research Center. Additionally, the code executed at 1.59 gigaflops on an eight-processor CRAY Y-MP at the Numerical Aerodynamic Simulation (NAS) facility at Ames Research Center. The code was submitted to Cray Research, Inc. (CRI) as an entry in the Gigaflop Performance Award Program, which recognized complete applications, from initialization through final output, that had a sustained performance of over 1 gigaflop. This application was the only application recognized that achieved the required performance on a CRAY-2.

The code models Taylor-Couette flow, which is induced by rotating one of two concentric cylinders with respect to the other. This type of flow has been studied for over a century as a model of the instability of curved and rotating shear flows. A critical rotational speed of the inner cylinder with respect to the outer cylinder causes regular-sized axisymmetric annular vortices to fill the entire gap between the cylinders. These Taylor vortices lose axial symmetry in regular azimuthal waves at higher speeds. Eventually, they show contained bursts of turbulence. The turbulence and chaotic nature of this flow are the primary reasons for the lengthy execution of the application. An existing production code (NSCY3D) that was developed at Langley Research Center (ref. 1) simulates these events in a three-dimensional, time-dependent flow; this code was the application chosen for this attempt at parallelization.

Section 2 of this report is a brief discussion of multitasking on CRAY computers. Section 3 describes the operating environment and the methodology used for analysis and verification. Sections 4 and 5 describe the single-processor and multiple-processor optimizations of the code. Timing results are given in section 6, and section 7 summarizes the work.

Appendixes A and B contain coding details for some of the parallelization effort.

2. Multitasking on CRAY Computers

Cray Research supports three methods of multitasking: macrotasking, microtasking, and autotasking (refs. 2 and 3). Macrotasking is done with calls to subroutines in a multitasking library, while microtasking and autotasking are both directive driven.

The original version of multitasking, which is Cray Research's name for the different methods of invoking parallel execution, was called macrotasking. As the name implies, macrotasking relies on relatively large-grain parallelization, generally at the subroutine level. An initial evaluation of macrotasking revealed both strengths and weaknesses. To spawn parallel tasks, the programmer must call CRAY macrotasking library subroutines. The analysis of data must be done on a subroutine basis to determine which variables are shared by all tasks and which are private to each individual task. This tedious modification leaves the code in a nonportable form, except to other CRAY multiple CPU (central processing unit) systems. On the positive side, programs that exhibit large-grain parallelism have very little overhead added to the total CPU time, and may, in a dedicated environment, have an elapsed time that is very close to the total CPU time divided by the number of available processors.

Task management of macrotasking is performed by the library scheduler, which gets information regarding synchronization from the subroutine calls (ref. 3). The library scheduler attaches tasks to logical CPU's or processes. The job scheduler then attaches the logical CPU's to physically available CPU's. Depending on the system load, parallel tasks may run concurrently. Macrotasking requires subroutines to manipulate tasks, to control critical regions, and to control events. Not all codes can benefit from using this method of multitasking. Performance may suffer when the true task granularity is small, when the number of available processors does not match the number of tasks that have been spawned, or when parallel task sizes are not balanced across processors.

Some work has been done (ref. 4) to provide a portable parallel programming language, FORCE, for a group of shared-memory multiple-instruction, multiple-data (MIMD) multiprocessors. This language uses the UNIX *sed* editor and a set of script files to replace FORCE parallel constructs embedded within a FORTRAN code with the appropriate machine-dependent multitasking subroutines. The programmer can then concentrate on the parallel aspects of the program without having to learn how to program in parallel on each machine. This tool has been useful to some researchers at Langley Research Center. (See ref. 5.)

The FORCE program became available at Langley at about the same time that Cray Research released microtasking. Microtasking was chosen over both FORCE and macrotasking for several reasons. The overhead for microtasking is lower than for macrotasking. On CRAY computers, FORCE uses macrotasking subroutines to control the parallelism. Microtasking a code does not automatically result in significant changes to the code. Compiler directives, which appear as comments to other machine compilers, are inserted to define the parallelism. Microtasking can efficiently parallelize tasks with small granularity (e.g., outermost DO loops). Since the programmer can look for parallelism below the subroutine level, analyzing data dependencies becomes much easier. In a FORCE program, parallelism must be expressed from the very beginning, not just in selected subroutines. In addition, microtasked programs behave much better than macrotasked programs in a batch environment.

The overhead for microtasking is much smaller than that for macrotasking, because microtasking does not require extensive task management. The microtasking preprocessor *premult* converts a user code into three routines for each original routine that is microtasked. The master copy, an assembly language routine, is called by the same name as the original. Its purpose is to decide whether the single-processor version or the multiprocessor version of the subroutine is to be called. This decision prevents any attempt to "nest" microtasking at run time. Parallelism is defined by control structures that designate segments of code that can be done by multiple CPU's in any order. The control structures are defined by the families of compiler directives CMIC$ DO GLOBAL and CMIC$ PROCESS. All work must be completed in a control structure before continuing, but this does not imply a barrier form of synchronization, where all processes must reach the control structure before any of them may proceed further. In other words, if the control structure work is done by a subset of the total number of processes, those active processes may proceed without waiting for the others.

Cray Research's latest multitasking product is called autotasking. Like microtasking, it is directive driven. The intention of the *cf77* FORTRAN compiling system is to analyze all data and insert autotasking directives automatically. Autotasking, however,

was in an early stage of development during the period that this work was performed. Therefore, it was not utilized for this study.

3. Environment, Analysis, and Verification

During the early phase of this study, the CRAY-2 at Langley was running UNICOS 4.0.9 with version 3.0 of the Cray FORTRAN compiler (*cft77*). The CRAY Y-MP at NAS was running a Beta UNICOS 5.0 with version 3.1 of the *cft77* compiler. Access to a dedicated machine was limited at both locations, so the majority of the development work was done in a heavily loaded nondedicated environment.

The first task was to analyze the code to determine which subroutines should receive the primary optimization effort and how to determine the efficiency of any optimizations that were attempted. Next, the code had to be optimized both for single- and multiple-processor executions. These efforts are described in detail in sections 4 and 5. Finally, it was necessary to verify that the microtasking had been implemented correctly.

The CRAY utility *flowtrace* provides an easy means for profiling a program without much overhead. As suspected, the most computationally intensive subroutines were matrix multiplications and FFT's. The matrix multiplications were performed by using the CRAY library subroutines TRANS and MXMA. The FFT's included in the original application code were FORTRAN implementations of Temperton algorithms (refs. 6 and 7). Although a functionally equivalent FFT, called RFFTMLT, had been available in the CRAY scientific library, it offered no improvement in performance. Subsequently, the Temperton algorithms were coded in CRAY assembly language (CAL) by Cray Research, who then replaced the previous version of RFFTMLT in the scientific library with the more efficient CAL implementation. The user interface to the revised FFT's remained compatible.

In a heavily loaded batch environment, the execution time of the single-processor version of the code varied by only 3 to 5 percent for identical runs on Langley's CRAY-2 and was essentially the same as for a dedicated machine. This was significantly better than the 20- to 30-percent uncertainty between execution times exhibited by the CRAY-2 at NAS. The primary difference between the two machines is the memory. The Langley CRAY-2 has static random-access memory (SRAM) with a clock cycle of 45 nsec, while the CRAY-2 at NAS has dynamic random-access memory (DRAM) with a clock cycle of 80 nsec and requires refreshing at each cycle. Thus, the SRAM CRAY-2 does not suffer as much adverse effect from memory conflicts between competing processes as the DRAM CRAY-2. Since there was a relatively small uncertainty in batch execution time between runs on the Langley CRAY-2, changes with a significant impact on the execution time (for the single-processor version of the code) in either direction should be observable, even on a nondedicated machine.

Since one goal was to achieve a computational rate of over 1 gigaflop, simply measuring the CPU utilization time of the program was obviously not going to give an accurate picture of the efficiency of microtasking. Another uncertainty was whether any of the optimizations would have a major impact on the number of floating-point operations per iteration. The CRAY-2 does not have any utility that is analogous to the hardware performance monitor *(hpm)* available on the CRAY Y-MP. For very little overhead, *hpm* produces a report that gives the total number of floating-point additions (subtractions), floating-point multiplications, and reciprocals.

The code was monitored frequently during the optimization process by using *hpm* on the NAS CRAY Y-MP to see if the number of operations had changed significantly through any of the optimizations. It was possible to determine how many floating-point operations were required for program initialization, for program shutdown, and per iteration. This information allowed the computation of the code's gigaflop performance, internal to program execution, in either single-processor mode using elapsed CPU time or multiple-processor mode in a dedicated environment using elapsed wall time.

Errors in a code being run in parallel are, in general, nondeterministic. To minimize the chance of introducing errors, modifications were made in very small steps. The standard formatted output file detected any gross errors. However, to find subtle errors, a check program was required. This check program compared the binary restart file from the original unmodified code with the binary restart file from each successive modification execution. Since there was no reason to expect exact agreement between successive runs, a relative error test was used to compare the two binary files. The only differences between runs was round-off error caused by a slightly different ordering of numerical calculations.

4. Single-Processor Optimization

The most important maxim in parallelizing a code on a vector processor is to not forsake vectorization

for parallelization, since the biggest relative performance improvement over scalar code arises from vectorization. Thus, the first thrust was to examine the execution of the single-processor version of the code. A test case of 25 iterations was chosen, because it was sized by execution time to run interactively rather than in batch mode. The code that was optimized, NSCY3D, was developed at Langley on the VPS-32, a modified CYBER 205. When the code was first developed, PARAMETER statements had not yet been added to the CYBER 200 FORTRAN language. In addition, the VPS-32 architecture favored long contiguous vectors (up to 65 535 elements), so many of the DO loops were collapsed from two or three levels down to one. When the code was translated to the CRAY-2, the grid size was also increased, so that many of these collapsed loops had iteration lengths of 137 280. These characteristics had a major impact on the optimization effort.

Without any modifications, NSCY3D executed at 220 million floating-point operations per second (megaflops) on the CRAY-2. On the CRAY Y-MP, it executed at 202 megaflops. Virtually every other program that was tested on the two machines ran between 1.5 and 2.5 times faster on the CRAY Y-MP. Using the *flowtrace* utility, this discrepancy was found in the scientific library routine MXMA, which was much more efficient on the CRAY-2, because of the CRAY-2's faster cycle time and the use of its fast local memory for storing intermediate results. The MXMA assembly code is written so that memory accesses to local and main memory and calculations in the addition and multiplication functional units can all occur simultaneously. The information from *flowtrace* confirmed that most of the program CPU cycles were spent in the FFT's and MXMA.

The program was also compiled with the loopmark compiler option *(-em)* turned on. In addition to marking DO loops so they can be easily found in a listing, the type of loop is also indicated. Loops may be scalar, vector, or short vector (less than 64 elements). The compiler identified virtually no short vector loops, even though the array dimensions were 33, 65, and 64 elements. When the PARAMETER statement became available, it had been used to dimension the arrays but not as limits on any DO loops. Since there was no information to tell the compiler how long each loop was, *cft77* was generating code to stripmine each DO loop. Since the vector registers on the CRAY machines hold only 64 elements each, vector computations must be done in chunks of 64 elements, which results in the compiler generating another loop (stripmining) around the vector loop for 64 elements. This loop is executed until all the elements in the original, longer vector have been processed.

The parameters NDR, NDZ, and NDY corresponded to the dimensions 33, 65, and 64 elements, respectively. The ranges on DO loops used the variables NR, NZ, and NY. Conceptually, changing the variables to parameters was a trivial modification to the code. However, making all the changes at once led to an editing error that gave incorrect results and forced a restart for the modification. Thus, the lesson of keeping modifications simple was reinforced. Making the changes in three stages was successful and resulted in a surprising improvement of almost 10 percent to a rate of 240 megaflops. In addition to improving the performance of the short vector loops, which made up only a small percentage of the code, using parameters improved the performance of other vector loops as well. This improvement resulted because the compiler had more information at compile time to precompute many of the offsets within all the DO loops and because of more efficient stripmining. An improvement in the range of the uncertainty of the timings was expected because of the use of parameters, so this enhanced performance was a surprise.

The information from *flowtrace* had already shown that most of the CPU time was being spent in the subroutines RMULT, ZMULT, and ZMULTH. Each of these routines was short and consisted only of a call to the scientific library routine MXMA and, in the case of RMULT, calls to TRANS, which computes a matrix transpose. Subroutine TRANS takes a significant amount of CPU time but has no floating-point operations. In this code, TRANS was used with MXMA to compute $\mathbf{Q} = \left(\mathbf{A}\mathbf{B}^T\right)^T$. This computation was accomplished by calling TRANS to compute $\mathbf{C} = \mathbf{B}^T$, followed by a call to MXMA to compute $\mathbf{W} = \mathbf{A}\mathbf{C}$, and, finally, another call to TRANS to compute $\mathbf{Q} = \mathbf{W}^T$. However, since an equivalent formulation is $\mathbf{Q} = \mathbf{B}\mathbf{A}^T$, only one transpose is necessary. Furthermore, MXMA has the capability to use the transpose of $\mathbf{A}$ as one of the source matrices, with the appropriate identification of stride between adjacent row and column elements. Consequently, both calls to TRANS within RMULT were eliminated. The code, as originally developed for the VPS-32, had required the call to the transpose routine because the VPS-32 needed vectors that had a unit stride through memory to be efficient. Therefore, since array storage is columnwise in FORTRAN, the transpose was necessary to make row operations into unit stride vector operations. Since RMULT, ZMULT, and ZMULTH now only called MXMA, these three

subroutines were eliminated, and the calls to MXMA were placed in-line. With the overhead of subroutine calls and the transpose removed, NSCY3D showed a speedup of about 5 percent, compared with the original version, and now ran at 250 megaflops. This performance improvement was verified with a run in dedicated time.

Several more changes were made to NSCY3D, none of which showed the improvements of the first changes. Even though these changes were minor for this code, except for the fact that without them a rate of 1 gigaflop would have been unattainable, they are described briefly to assist others with codes of different designs.

The binary input-output (I/O) in NSCY3D was done within a DO loop and with an implied DO structure. The loopmark compiler option gave somewhat misleading information in this case. It indicated that the I/O had been vectorized, which might cause a user to think that the I/O was being done as efficiently as possible. The I/O was restructured to write an entire array without any DO structure. The I/O time was reduced by over 90 percent, which translated into an overall performance improvement of 4 to 5 megaflops.

There is no divide functional unit in the CRAY architectures. Division is done through reciprocal approximations, which makes it the most expensive floating-point operation. Within the subroutines EQ1, EQ2, and EQ3, there were repeated reciprocal calculations that used the invariant array R to compute R inverse and the inverse of R squared. These calculations were moved to the main program, were done once, and were stored in a COMMON block. Several DO loops in the main program and in subroutine SETUP were interchanged, which transformed an inner-product calculation to an outer-product calculation. The subroutine HCYLSLD was called at two places in NSCY3D. One of the calls was from within a DO loop. Another version of HCYLSLD was created, for which the DO loop was pulled into the routine itself. The net result of all these minor changes was an increase of about 10 megaflops.

The FFT's calculated by the FORTRAN subroutine FFT991 and the subroutines that it called were identical to the transforms calculated by the scientific library routine RFFTMLT. All calls to FFT991 were replaced by calls to RFFTMLT. Additionally, all the extra FORTRAN coded routines called by FFT991 that had equivalent optimized CAL versions in the scientific library were removed. After this final modification, the code executed at 272 megaflops in a dedicated environment on the CRAY-2. The *hpm* utility indicated that the only significant change in operation count was for floating-point reciprocals. Before all these optimizations, there were about 120 000 reciprocals per iteration, and afterward there were about 20 000 per iteration. However, since the total operation count was about 1.32 billion per iteration, the reduction was only significant in the execution time. Also, the code then executed at 223 megaflops on the CRAY Y-MP. Achieving a gigaflop on the eight-processor CRAY Y-MP seemed certain, but it seemed unlikely to exceed about 0.9 gigaflop on the four-processor CRAY-2.

Not one of the changes made to the single-processor version was conceptually very complicated. For this code, the most obvious optimizations had the largest impact on performance. They have been described to emphasize that many older codes that were efficient when written for other versions of FORTRAN and different architectures could benefit from some fine-tuning for CRAY computers. It was also interesting to note the magnitude of performance improvement that resulted. Since the goal was to obtain the absolute best single-processor performance, the effort expended far exceeded the effort the typical programmer would need or want to expend. A single-processor rate in the mid to upper 260-megaflop range could have been achieved with far less effort; however, the gigaflop goal would not have been possible. From a purely practical standpoint, running at 1.01 gigaflops is little different from running at 900 megaflops. It is not recommended that anyone try to squeeze as much performance as was done for this study, since the performance payoff for effort expended drops dramatically near the end of the optimization effort.

5. Microtasking the Code

The code that was first analyzed for microtasking did not have any of the single-processor optimizations discussed in section 4. As the other changes to the single-processor version were tested and verified, they were added to the version being microtasked. The single-processor version that was first completely microtasked and timed had the DO loop ranges specified by parameters, the calls to TRANS eliminated, the calls to MXMA in-lined, the binary I/O reformulated, and the inner products replaced by outer products. This section details much of the microtasking effort to emphasize the relative ease with which much of the code was microtasked with only a few compiler directives. Some examples are given to illustrate the technique that was used to microtask subroutine calls when the source code was not available for manual insertion of microtasking directives.

5.1. Data Analysis

The flow trace analysis was used to determine which routines were the most time-consuming. These routines were microtasked first. After selecting a subroutine to microtask, the data were analyzed to determine if variables were shared or private. Shared variables are defined by one memory location; therefore, an update to a shared variable by one CPU is known to all other CPU's. On the other hand, each CPU has a separate storage location for private variables. Variables that appear in the argument list, COMMON blocks, DATA, or SAVE statements must be treated as shared. All remaining variables must be treated as private. In general, shared data are modified within microtasking control structures and private data are modified outside of control structures. In some subroutines, several arrays used as work arrays were removed from the subroutine argument list. By declaring the arrays as local variables, the arrays were converted from shared to private. To exploit the parallelism in these subroutines, all CPU's needed separate copies of the work arrays. The maximum amount of space required by each CPU for the work arrays was approximately 2 * NDZ * NDZ (i.e., <9000 words).

The original main program had several sections of code that were good candidates for microtasking. Since microtasking is not allowed in the main program, two subroutines were created to hold these sections so that their parallelism could be exploited.

5.2. Microtasking Directives

The following microtasking directives were used in this application:

```
CMIC$ DO GLOBAL
CMIC$ DO GLOBAL FOR NCPU
CMIC$ PROCESS/CMIC$
      ALSO PROCESS/CMIC$ END PROCESS
CMIC$ CONTINUE
```

These directives alone defined the processor flow through each subroutine.

Since microtasking is usually invoked at the DO loop level, probably the most frequently used microtasking compiler directive is CMIC$ DO GLOBAL. This control-structure directive is placed before an outer DO loop whose iterations do not depend on the results of any other iteration. Each iteration of the loop is executed by CPU's as they become available. For example, if the system is heavily loaded, one or two CPU's may do most of the work.

As mentioned previously, one of the characteristics of this code is that a number of nested DO loops had been collapsed into single loops. A form often used was

```
DO 100 IJK = 1, NDR * NDZ * NDY
```

This loop can be executed in parallel on multiple CPU's by placing the following directive directly before the loop:

```
CMIC$ DO GLOBAL FOR NCPU
```

where NCPU is the number of parallel tasks. NCPU was set to 4 on the CRAY-2 and to 8 on the CRAY Y-MP. This directive divides the total number of iterations into NCPU parts, each of which can be executed by a separate CPU. The variable NCPU was declared in the statement

```
PARAMETER (NCPU = 4)
```

which was placed in the main program and in each microtasked subroutine. Single loops may be efficiently multitasked if the number of iterations is sufficiently large. Multitasking loops with small numbers of iterations and a small amount of computation may lead to negligible speedups.

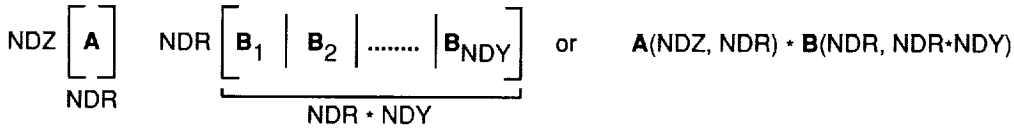
The CMIC$ PROCESS and CMIC$ END PROCESS directives define a control structure and were used together to enclose segments of code which were to be done by one processor. The trio CMIC$ PROCESS, CMIC$ ALSO PROCESS, and CMIC$ END PROCESS were used to define two sections of code that were to be executed in parallel.

Finally, the CMIC$ CONTINUE directive was placed before calls to subroutines that contained microtasking directives when the calling subroutine was itself a microtasked subroutine. Without the directive, the preprocessor would place a call to the single-processor version of the routine, rather than to the multiple-processor version.

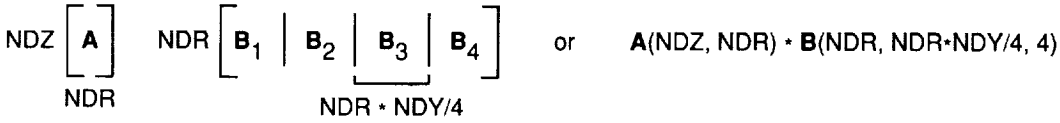
5.3. Exploiting Parallelism

The parallelism to be exploited through microtasking was present in several different forms. The most significant forms are discussed below. Data structure is always a key factor in exploiting parallelism. Most of the arrays involved in the description in this section are of the form (I1, I2, NDY), where I1 and I2 are some combination of NDZ and NDR.

(a) Single processor, Kth matrix multiply.



(b) Optimized for a single processor.



(c) Multiprocessor version (NCPU = 4).

Figure 1. Equivalent methods to formulate matrix multiplications.

5.3.1. Nested Loops

Many of the DO loops were of the form

```
DO K = 1, NDY
BODY (I, J, K)
```

The body of code within the K loop typically consisted of a FORTRAN source that was a doubly nested loop over the first two dimensions. In many cases, it included a call to MXMA to multiply two matrices together prior to entering the double loop. One or both of the matrices were typically just the Kth plane of a three-dimensional array. Consequently, placing

```
CMIC$ DO GLOBAL
```

prior to the DO K loop identified NDY-independent (and usually large-grained) vectorized tasks to be done in parallel.

5.3.2. Single Long Loop

The original code was optimized for the CYBER 205 architecture in many places by collapsing triply nested loops into a single, long loop as follows:

```
DO IJK = 1, NDR * NDZ * NDY
BODY (IJK)
```

Although it would have been possible to return to the original triply nested form, it was easier and more efficient to insert

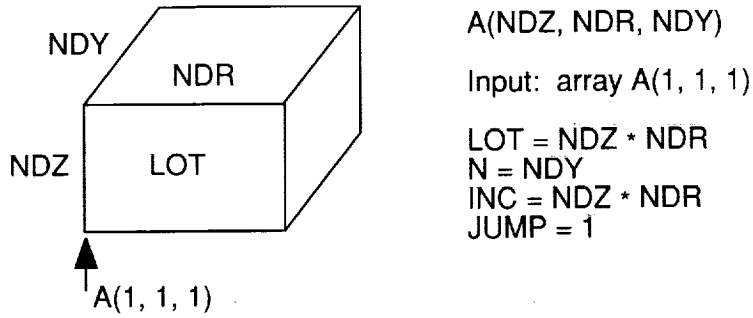
```
CMIC$ DO GLOBAL FOR NCPU
```

before the DO IJK loop. Parallelization takes place through an automated partitioning of the loop length into NCPU parts. Vectors are still long at a nominal size of 137 280/NCPU for the problem size of interest.

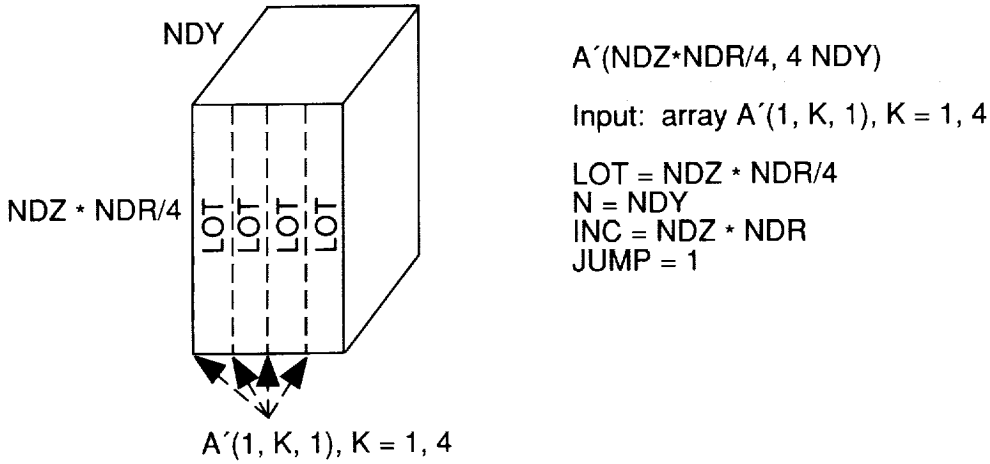
5.3.3. Single Calls to Scientific Library Subroutines

Since most of the execution time was spent in the matrix multiplication and fast Fourier transform subroutines, it was imperative to achieve top efficiency in these two areas. In some of the calls to MXMA and in the calls to RFFTMLT, there was but a single call to the subroutine. Hence, to define independent, parallelizable tasks, it was necessary to partition the calculation and the arrays involved into independent parts.

5.3.3.1. Single MXMA calls. One of the calls to MXMA was the result of previous optimization, which enabled multiple matrix multiplications to be performed with a single call to MXMA. Here, as illustrated in figure 1, the array B(NDR, NDR, NDY) is viewed as a single matrix B(NDR, NDR * NDY).



(a) Single processor.



(b) Multiple processors (NCPU = 4).

Figure 2. Fast Fourier transform data structure.

This view is mathematically correct because the left matrix in each of the matrix multiplications is the same. The single-processor efficiency is increased because the MFLOP rate for MXMA increases as the number of columns of the second matrix increases. To parallelize this subroutine, another step is necessary. Consider that B is also equivalent to an array B(NDR, NDR * NDY/NCPU, NCPU). Viewed this way, the problem is partitioned into NCPU-independent matrix multiplications. An outer DO loop from 1 to NCPU with pointers to the appropriate columns in B can now be parallelized with a CMIC$ DO GLOBAL directive.

5.3.3.2. FFT calculations. As previously mentioned, the original FFT calculations were calculated with the FORTRAN subroutine FFT991 and the subroutines it called. The FORTRAN FFT's were replaced with the functionally identical scientific library subroutine RFFTMLT, which resulted in superior performance. Subroutines FFT991 and RFFTMLT have identical calling sequences. The argument list contains the name of the array to be transformed, the length of each transform (N), the increment in storage between successive elements of the vector to be transformed (INC), the increment in storage between the starting elements of each data vector (JUMP), and the number of independent vectors to be transformed (LOT).

The initial parallelization effort was to microtask the FFT991 DO loops at the source level by dividing the loops over the LOT transforms into NCPU parts. This approach was not very successful, because most of the loops were innermost loops. Therefore, there was a lot of computation in outer loops that was not parallelized. Efforts to parallelize at the outer loops were not successful because of load-balancing issues.

The parallelization of RFFTMLT required an alternate approach because the source code was unavailable. It was again necessary to conceptually view the computation and the data base as

different, but independent, NCPU problems. Unlike the MXMA partition where the data array was simply partitioned into smaller, but contiguous, NCPU segments, the data structure for the FFT is more complex. The data associated with each of the NCPU tasks are not contiguous, even though they were for the single-processor version. Fortunately, the RFFTMLT software was written with enough flexibility to permit the type of data structure needed. Figure 2 shows the data structure for both the single- and multiple-processor approaches.

For both the MXMA and RFFTMLT partitioning described in the preceding paragraph, the discussion is presented as if the array sizes were such that NCPU divided the dimensions exactly. Since that could not be assumed, the actual code leaves the arrays in their original form and computes pointers to the appropriate portions of the arrays. This is done in the initialization stage of the program.

6. Timing and Performance

This section presents results for several versions of the code. It was important to microtask as much as possible, since any nonparallel code has a strong impact on the speedup. For a five-iteration run, 97 percent to 98 percent of the execution time is spent in microtasked subroutines. Amdahl's Law predicts a theoretical maximum speedup of 3.67 to 3.77 for this percentage of parallel code. However, as the number of time steps is increased, the initialization code required becomes less significant in the overall execution time; as a result, the speedup should increase. Typically, 1000 time steps are required for this applications code.

6.1. Dedicated Environment

Tables 1 and 2 present results from two versions of the code that were executed in a dedicated environment for 50 iterations. The difference between the codes is in the implementation of the fast Fourier transforms. Table 1 contains the results for when the FORTRAN FFT's are used.

Table 1. CRAY-2 Dedicated Results for 50 Iterations With FORTRAN Subroutine FFT991

	1 CPU	4 CPU's
Elapsed seconds	256.0	71.9
CPU seconds	255.3	279.7
MFLOPS	258.0	921.4

The speedup for this code was approximately 3.6. The CPU overhead introduced by microtasking in this case is about 9.6 percent of the CPU time for the sequential code. The microtasking overhead is defined as the difference between the total CPU time for the microtasked code and the single-processor code divided by the total CPU time for the single-processor code. Overhead may result for several reasons, including overhead from the calls to the microtasking subroutines, the synchronization of tasks, load imbalance, memory contention, and time spent in sequential portions of the code where processors were connected but not used. Table 2 shows the results for when the scientific library subroutine RFFTMLT (appendix B) is used.

Table 2. CRAY-2 Dedicated Results for 50 Iterations With Scientific Library Subroutine RFFTMLT

	1 CPU	4 CPU's
Elapsed seconds	244.6	66.2
CPU seconds	243.4	257.5
MFLOPS	270.1	999.6

In this case, the speedup is 3.69, and the overhead due to microtasking is approximately 5.8 percent. The microtasking overhead decreased because of an increase in task granularity and because of more evenly balanced tasks by parallelizing on the subroutine calls. This decrease in overhead was responsible for the greater than expected increase in the MFLOPS from subroutine FFT991 to subroutine RFFTMLT.

Before making the final dedicated timings, a few additional changes were made. All DO GLOBAL directives, which preceded the outermost DO loops, were replaced by DO GLOBAL FOR NCPU directives. This change caused the total number of loop iterations to be divided into NCPU groups, one group for each processor; thus, the iterations are not passed out one at a time. Several loops, which were previously placed within the PROCESS and END PROCESS directives, were rewritten so that the iterations could be executed in parallel. In previous versions, they were left unmodified because they looked like they were more trouble than they were worth. Finally, the DO loop around the subroutine HCYLSLD was brought inside the subroutine. Tables 3 and 4 show the final results after these modifications. The timings obtained on the CRAY-2 for 100 iterations are shown in table 3.

Table 3. CRAY-2 Final Dedicated Results for 100 Iterations With 4 CPU's

Elapsed seconds	130.8
CPU seconds	512.3
MFLOPS	1011.6

Results from the CRAY Y-MP are shown in table 4.

Table 4. CRAY Y-MP Final Dedicated Results for 100 Iterations

	4 CPU's	8 CPU's
Elapsed seconds	156.9	83.3
CPU seconds	596.4	600.1
MFLOPS	844.0	1586.4

6.2. Batch-Environment Observations

Because of system load, any quantitative study of the effectiveness of multitasking must be done in a dedicated environment. Cray Research warns the user that macrotasking should not be used in a batch environment, since elapsed time can vary significantly from run to run depending on system load. Theoretically, microtasking can improve system throughput, because these jobs can efficiently use CPU's that become idle for a short period of time. However, this is true only on systems that are not heavily loaded. The CRAY-2 at Langley is generally running at over 99 percent of CPU utilization.

In the UNICOS 4.0.9 environment, macrotasked programs generally showed a marked decrease in overall throughput and a marked increase in CPU time. Microtasking jobs also showed a significant increase in CPU time. Under UNICOS 5.1.6, microtasked programs have CPU times that are slightly greater than the CPU times for the equivalent single-processor code.

Other factors may also affect the performance of a microtasked code in a batch environment (personal communication with S. K. Duggirala currently with Thinking Machines Corporation). The operational characteristics of the multitasking libraries are supposed to be transparent to the user. There can be significant idle (but connected) CPU time that is charged to the user.

The time slice and swapping of executing processes are also assumed to have an impact, though undetermined, on multitasked code. Duggirala further stated that if a user submits a microtasked job to a batch-job mix that is dominated by sequential code, substantial elapsed time speedup is observed. On the other hand, submitting a microtasked code to a batch-job mix that is dominated by multitasked codes yields little or no speedup. The environment on the CRAY-2, though overwhelmingly sequential, behaves more like a job mix that is dominated by multitasked code. This behavior is caused by the configuration of the batch queues and the large number of users at Langley Research Center. In general, 8 to 12 codes are candidates for execution at any time, so there are few spare CPU cycles of which a microtasked code can take advantage.

7. Concluding Remarks

When this project was started, it was believed that to achieve a sustained computational rate of over 1 gigaflop would require that the code run at not less than 300 megaflops on a single central processing unit (CPU) with about 95 percent parallelization. When the single-processor execution rate reached about 272 megaflops, it appeared that a peak rate of just under 1 gigaflop would be the best that could be accomplished on the CRAY-2. The final result of 1.01 gigaflops translates to a speedup of about 3.7, which indicated from Amdahl's Law that at least 97 percent of the code was being executed in parallel.

Some of the simple optimizations to the single-processor version had the greatest impact, which showed that user intervention in the multitasking process is important. Computer codes that have been converted from different architectures may have inherent inefficient constructs arising from the straight translation, where little consideration is given to the new architecture. For example, the calls to TRANS from within RMULT, prior to calling MXMA, arise from the CYBER 205 genesis of the code.

Although this project was time-consuming because of a basic unfamiliarity with microtasking, it demonstrated that once programmers are familiar with parallelization on CRAY computers, significant improvement in performance should be possible with minimal to moderate effort, depending on the structure of the code. Though final analysis of the effectiveness of any multitasking effort must still be done on a dedicated machine, preliminary microtasking versions can now be run in a nondedicated environment without incurring substantial execution-time penalty.

NASA Langley Research Center
Hampton, VA 23665-5225
September 5, 1991

Appendix A

Microtasking MXMA

A large percentage of the execution time is in the library subroutine MXMA, which is called from a number of locations in the code. In most cases, MXMA is called from within a single or nested DO loop, where the loop index variable appears as a subscript in the argument list of MXMA. In both cases, the iterations of the DO loops are independent of one another, so that calls to MXMA can be made simultaneously by multiple CPU's. In several other places in the code, a call to MXMA does not appear in the loop. In this situation, the work to be done by MXMA can be divided into NCPU parts. An example of a call to MXMA is as follows:

```
      CALL MXMA (A, NA, IAD, B, NB, IBD, C,
     $   NC, ICD, NAR, NAC, NBC)
```

where the arguments have the following meanings:

A	first matrix of the product
NA	spacing between column elements of A
IAD	spacing between row elements of A
B	second matrix of the product
NB	spacing between column elements of B
IBD	spacing between row elements of B
C	output matrix
NC	spacing between column elements of C
ICD	spacing between row elements of C.
NAR	number of rows in the first operand and result
NAC	number of columns in the first operand and number of rows in the second
NBC	number of columns in the second operand and result

There were two situations in which the work in MXMA was divided. In the first example, the original code was

```
      NW = NDR * NDZ
      CALL MXMA (ZOP1V, 1, NZ, W, 1, NZ, RH2,
     $   1, NZ, NZ, NZ, NW)
```

The modified code is

```
CMIC$ DO GLOBAL
      DO 10 K = 1, NCPU
      IP = ISTNW(K)
      CALL MXMA(ZOP1V, 1, NZ, W(1, IP),
     $   1, NZ, RH2(1, IP), 1, NZ, NZ, NZ,
     $   NWPTS(K))
   10 CONTINUE
```

In the example above, RH2 is the resultant matrix. Each processor multiplies matrix ZOP1V by one fourth of the matrix W to result in one fourth of the matrix RH2. The partitioned matrices are divided by columns, so that the elements in each processor's submatrix are stored contiguously.

The arrays NWPTS and ISTNW determine the amount and starting location of the work to be done in MXMA by each processor. The variables are calculated in the main program with the following code:

```
      NRNP = NR/NCPU
      NW = NDR * NDY
      NWNP = NW/NCPU
      DO 1 I = 1,NCPU
      NRPTS(I) = NRNP
      NWPTS(I) = NWNP
    1 CONTINUE
      IREM = NR - (NRNP * NCPU)
      IF (IREM.GT.0) THEN
      DO 2 I = 1,IREM
      NRPTS(I) = NRPTS(I) + 1
    2 CONTINUE
      ENDIF
      IREM = NW - (NWNP * NCPU)
      IF (IREM.GT.0) THEN
      DO 3 I = 1,IREM
      NWPTS(I) = NWPTS(I) + 1
    3 CONTINUE
      ENDIF
      ISTNR(1) = 1
      ISTNW(1) = 1
      DO 4 I = 2, NCPU
      ISTNR(I) = ISTNR(I - 1) + NRPTS(I - 1)
      ISTNW(I) = ISTNW(I - 1) + NWPTS(I - 1)
    4 CONTINUE
```

The arrays NRPTS and ISTNR were used with other MXMA calls.

In the second example, the original code was

```
      CALL MXMA(W2, 1, NZ, REVIV(1, 1, KT),
     $  NR, 1, W1, 1, NZ, NZ, NZ, NR)
```

The microtasked code is

```
CMIC$ DO GLOBAL
      DO 10 K = 1, NCPU
      IP = ISTNR(K)
      CALL MXMA(W2, 1, NZ,
     $  REVIV(IP, 1, KT), NR,
     $  1, W1(1, IP), 1, NZ, NZ, NZ,
     $  NRPTS(K))
   10 CONTINUE
```

In this example, matrix W2 is multiplied by the transpose of matrix REVIV, which yields matrix W1. Matrix REVIV is divided into four parts along the rows. The resultant submatrix W1 is divided along the columns, so that the elements in each processor's W1 are contiguously stored.

Appendix B
Microtasking the FFT's

As mentioned previously, the original program did not use the scientific library subroutines for calculating the fast Fourier transforms. The FORTRAN subroutines involved in the FFT's consisted of FFT991, FFT99A, FFT99B, VPASSM, FFTFAX, FAX, and FFTRIG. The first four subroutines calculate the transforms, while the other three calculate the sines, cosines, and factors needed for computing the FFT's. In the original program, a call similar to the following is made:

```
      CALL FFT991(WC1, WC2, TRIGS, IFAX,
     $  INC, JUMP, N, LOT, ISIGN)
```

In turn, FFT991 calls the other lower-level routines. The array WC1 holds the data vectors that contain strides of INC and length N, while WC2 is used as work space. The arrays TRIGS and IFAX have been initialized previously. The number of data vectors to be transformed is LOT, and the distance between the starting elements of each vector is JUMP.

The microtasking approach taken was to divide the loops that ranged from 1 to LOT into four pieces, one for each processor. For the majority of the execution time, LOT is equal to 2145; at other times, LOT is equal to 196. Each loop was preceded by the directive

```
CMIC$ DO GLOBAL FOR NCPU
```

In several cases, these loops were the outermost loops of a nested DO loop, but in most cases, they were the innermost loops, where the outer loops ranged from 1 to LA and where LA was equal to 1, 2, or 8. Clearly, microtasking over this outer loop would have resulted in poor load balancing. When necessary, the assignment statements of additional variables involved with indexing in these loops were rewritten so that they were independently calculated across iterations.

In later versions of the code, calls to FFT991 were replaced with calls to the scientific library subroutine RFFTMLT, which required the same argument list. Subroutine RFFTMLT is also microtasked by dividing the set of vectors to be transformed across the four CPU's. For example,

```
CMIC$  DO GLOBAL
       DO 10 J = 1, NCPU
       CALL RFFTMLT(WC1(IFWA(J), 1),
      $  WC2(1, J), TRIGS, IFAX, INC, JUMP,
      $  N, LOTR(J), ISIGN)
   10  CONTINUE
```

The array element LOTR(J) designates the number of vectors to be transformed, and IFWA(J) indicates the starting point for the Jth processor. Array WC2 is redimensioned so that each processor gets one fourth of the work space. The following code calculates these two arrays in the main program:

```
      IREGPTS = LOT/NCPU
      IREM = LOT - (IREGPTS * NCPU)
      DO 1 J = 1, NCPU
    1 LOTR(J) = IREGPTS
      IF (IREM.GT.0) THEN
      DO 2 J = 1, IREM
    2 LOTR(J) = LOTR(J) + 1
      ENDIF
      IFWA(1) = 1
      IF (NCPU.GT.1) THEN
      DO 3 J = 2, NCPU
    3 IFWA(J) = IFWA(J - 1) + LOTR(J - 1)
      ENDIF
```

8. References

1. Streett, C. L.; and Hussaini, M. Y.: A Numerical Simulation of the Appearance of Chaos in Finite Length Taylor-Couette Flow. *Appl. Numer. Math.*, vol. 7, no. 1, Jan. 1991, pp. 41–71.

2. *CFT77*™ *Reference Manual.* SR-0018 C, Cray Research, Inc., c.1988.

3. *CRAY-2*™ *Multitasking Programmer's Manual.* SN-2026 B, Cray Research, Inc., c.1988.

4. Jordan, Harry F.; Benten, Muhammad S.; Arenstorf, Norbert S.; and Ramanan, Aruna V.: *Force User's Manual – A Portable, Parallel FORTRAN.* NASA CR-4265, 1990.

5. Agarwal, Tarun K.; Storaasli, Olaf O.; and Nguyen, Duc T.: A Parallel-Vector Algorithm for Rapid Structural Analysis on High-Performance Computers. *A Collection of Technical Papers, Part 2—AIAA/ASME/ASCE/AHS/ASC 31st Structures, Structural Dynamics and Materials Conference,* Apr. 1990, pp. 662 672. (Available as AIAA-90-1149-CP.)

6. Temperton, Clive: Self-Sorting Mixed-Radix Fast Fourier Transforms. *J. Comput. Phys.*, vol. 52, no. 1, Oct. 1983, pp. 1 23.

7. Temperton, Clive: Fast Mixed-Radix Real Fourier Transforms. *J. Comput. Phys.*, vol. 52, no. 2, Nov. 1983, pp. 340–350.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY(Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1991	Technical Memorandum

4. TITLE AND SUBTITLE
Gigaflop Performance on a CRAY-2: Multitasking a Computational Fluid Dynamics Application

5. FUNDING NUMBERS
WU 505-90-52-02

6. AUTHOR(S)
Geoffrey M. Tennille, Andrea L. Overman, Jules J. Lambiotte, and Craig L. Streett

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23665-5225

8. PERFORMING ORGANIZATION REPORT NUMBER
L-16932

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TM-4305

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper describes the methodology for converting a large, long-running applications code that executed on a single processor of a CRAY-2 supercomputer to a version that executed efficiently on multiple processors. Although the conversion of every application is different, a discussion of the types of modifications used to achieve gigaflop performance is included to assist others in the parallelization of applications for CRAY computers, especially those that were developed for other computers. An existing application, from the discipline of computational fluid dynamics, that had utilized over 2000 hours of CPU (central processing unit) time on a CRAY-2 during the previous year was chosen as a test case to study the effectiveness of multitasking on a CRAY-2. The nature of the dominant calculations within the application indicated that a sustained computational rate of 1 billion floating-point operations per second, or 1 gigaflop, might be achievable. The code was first analyzed and modified for optimal performance on a single processor in a batch environment. After optimal performance on a single CPU was achieved, the code was modified to use multiple processors in a dedicated environment. The results of these two efforts were merged into a single code that had a sustained computational rate of over 1 gigaflop on a CRAY-2. Timings and analysis of performance are given for both single- and multiple-processor runs.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Microtasking; Multitasking; Parallelization	15
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298(Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

NASA-Langley, 1991